The Open Master Hearing Aid (openMHA)

4.17.0

Matlab Coder integration

The Open Master Hearing Aid (openMHA) – Matlab Coder integration

HörTech gGmbH
Marie-Curie-Str. 2
D–26129 Oldenburg

Contents

1 Introduction

For many audiological researchers the tool of choice for prototyping new algorithms is MATLAB. When the prototype reaches a certain stage of maturity there is oftentimes the desire to test the new algorithm within the context of a quasi realistic real-time hearing aid processing and/or under field conditions embedded in a mobile processing platform.
openMHA offers the researcher a powerful and flexible toolset capable of real-time audio processing even on limited hardware, but it is written in C++. Porting an advanced signal processing algorithm from MATLAB code to C++ can be a hassle and sometimes poses insurmountable due to limited manpower or institutional knowledge of C++.
This document describes how to integrate user MATLAB code into openMHA as a plugin via translation to C/C++ by the MATLAB Coder.

Nomenclature

- **user code** refers to the MATLAB code the user wants to integrate into openMHA via MATLAB Coder,
- **user function** means the entry point functions in the user code and their translated forms,
- **generated code** refers to the C/C++ source code the Coder generates from this code,
- **user library** refers to the shared library compiled from the generated code.
- Text written like `this` refers to names of variables or structs in source code and `call()` refers to functions. `this.m` means file names.

Prerequisites

In order to make use of this document the user needs a copy of openMHA, either in source code or binary form, a MATLAB Coder license and a general understanding of the usage of openMHA. In order to use the matlab_wrapper plugin the generated code needs to be compiled either from within the MATLAB Coder or manually. See the MATLAB Coder documentation on how to integrate a compiler into the Coder. The user should have some idea on how to answer the following questions:

- What the purpose of a compiler?
- What is the difference between source code and compiled code?
- What is a plugin in the openMHA context?
- What is a MATLAB struct?

For more information consult the openMHA application manual.

Document structure

There are two ways to integrate the generated code into openMHA as as plugin: The matlab_wrapper plugin and the ‘native compilation’. The matlab_wrapper plugin is easier to use but less flexible. The plugin is provided the name of the user library to the plugin. The plugin then loads the user library and calls the user functions at the appropriate times. This approach relies on the user code following a prescribed form described in section 3.

The ‘native compilation’ approach offers more flexibility but the user must be able to set up a development environment able to compile openMHA from source, and know some C++ in order to integrate the generated code into the provided source code. This approach is described in section 4.

Which approach to use?

There is no hard and fast rule on which approach to use for a given algorithm. The following guidelines can be used to figure out which approach probably fits best.
Usage of the matlab_wrapper plugin is recommended if:

- There is little or no institutional knowledge of C++
- The user code does hold no or only simple state, i.e. there is no or only simple data that needs to be stored from one audio block processing to the next.
- No or little configuration at runtime is needed
- The user code has a monolithic structure, i.e. it can be thought of as one big black box where the signal goes in and output comes out
- Little or no interaction with the rest of openMHA is desired

On the other hand the native compilation approach should be used if:

- Data sharing beyond the audio signal itself with openMHA is needed
- The algorithm structure itself is subject to change
- The user code is modular and the modularity needs to be preserved
- It is impossible to rewrite the user code to the prescribed structure for the wrapper plugin

Independent of the approach the integration will be easier if the code already fits the structure described in 3.1. If it is known at the beginning that an integration into openMHA is desired it can be advantageous to write the user code according to the described structure in the first place.

2 The MATLAB Coder in a nutshell

This section only introduces the most important terms needed to understand this documentation. It can not replace the Matlab Coder documentation. Please consult the official Matlab Coder manual for further information.

Introduction

The Matlab Coder generates C/C++ code from MATLAB code. This code can then be compiled using the MATLAB compiler or any other compiler. The generated code can be integrated into openMHA either in source code or in compiled form via the matlab_wrapper plugin. The matlab_wrapper plugin can only accept compiled C code.

Entry-point functions

An entry-point function is a top-level MATLAB function that gets compiled to C/C++ code. Only functions marked as entry-point functions are guaranteed to be generated as callable functions visible from the outside of the user code. The matlab_wrapper plugin expects some entry-point functions to be present, see subsection 3.1.

Input types

Because C is statically typed, all input and output types must be known at compile time. Unlike in MATLAB, input and output types become part of the function signature and can not be changed later, including array dimensions. If a function needs to accept variable size arrays, they need to be wrapped in a struct, done automatically at code generation.
The MATLAB Coder handles double, single, and half precision floating point numbers, 8-, 16-, 32-, and 64-bit signed and unsigned integers, logicals (booleans), characters and structs, cell arrays and strings. These can be either single values or matrices. The size of a matrix is denoted by `X` × `Y`, `X` denoting the number of rows and `Y` the number of columns. `:X` means 'up to X rows/columns' and `Inf` means an indeterminate size. The type of input argument can either be specified manually or defined by example. See the MATLAB Coder documentation for details.

3 Usage of the matlab_wrapper plugin

The matlab_wrapper plugin is the easiest but most restricted way to integrate MATLAB code into openMHA. To use it, the user compiles the user code into a shared library. The matlab_wrapper plugin then takes the library name without suffix as configuration variable `library_name`. The plugin then automatically resolves the entry-point functions and calls them during the appropriate callback, passing signal dimensions and input signal to the user code. Because the functions are resolved by name, the user code has to follow the form described in 3.1 for the matlab_wrapper plugin to properly resolve them.
If configuration at run time is desired, the `user_config` struct can be used (see section 3.1 for details). The plugin parses the entries of user configuration and creates an openMHA configuration variable for each entry. These variables can be changed during processing without impeding real-time safety.

3.1 Writing code targeting the matlab_wrapper plugin

3.1.1 User code structure

The user code and the plugin interface via four entry point functions: `init()`, `prepare()`, `process()`, and `release()`, of which `process()` is mandatory. These functions are automatically called by the wrapper plugin at construction, and during the `prepare()`, `process()` and `release()` callbacks respectively. In order for the matlab_wrapper plugin to properly resolve these functions, they must confirm to the prescribed interface, i.e. their input and output parameters must be exactly as described in the following.

`init()` is called when the user library is loaded. It must follow the form:

```
1 function [user_config,state] = init(user_config,state)
```

`user_config` is a `1xInf` array of structs containing the following members:

name A 1xInf char array, the name of

value An InfxInf doubles array

`state` has the same type as `user_config`. If user defined configuration variables are desired, `user_config` must be created within the `init()` function, e.g.:

```
1 function [user_config,state] = init(user_config,state)
2 user_config =...
```

```
3 [struct('name','writeable', 'value',ones(1,1))];
4   end
```

The size of `user_config` may not be changed after the call to `init()`. If the values of the elements of `user_config` need to be changed depending on the signal dimensions, this can be done during the prepare call, where `signal_dimensions` and `user_config` are available. The `state` variable is initialized in the same way and used to keep state data in between calls to `process()`.

```
1   function [user_config,state] = init(user_config,state)
2   state = [struct('name','rmslevel','value',ones(1,1))];
3   end
```

`prepare()` is called when the prepare command is issued. All initialization that depends on the form of the signal should be done here, furthermore the properties of the input signal can be checked, i.e., in case the processing requires a fixed number of channels or a certain sampling rate is required. If the processing changes any signal property, the appropriate member of `signal_dimensions` must be changed accordingly to inform the openMHA framework of the change. `prepare()` takes two arguments:

signal_dimensions is a struct with information about the input signal. If the processing changes any of the following parameters, they must be changed in the prepare call accordingly:

channels A uint32 containing the number of channels in the signal.

domain A char containing either 'W' for waveform domain or 'S' for spectral domain.

fragsize A uint32 containing the fragment size.

wndlen A uint32 containing the window length of the FFT if in spectral domain, zero otherwise

fftlen A uint32 containing the Length of the FFT in in spectral domain, zero otherwise.

srate A double containing the sampling frequency of the signal.

user_config as above.

state as above.

Example:

```
1   function [signal_dimensions, user_config, state]=...
2            prepare(signal_dimensions, user_config,state )
3   user_config(1).value(1,1)=2; % Assign a value to the
4                                % first element of user_config
5
6   % Store the rmslevel per channel
7   state(1).value=zeros(signal_dimensions.channels);
8
9   % Output contains only one channel
10  signal_dimensions.channels=1;
11  end
```

All signal processing has to happen in the `process_xy()` function. 'xy' takes different values depending on the input and output signal domains. Use 'ww' for waveform to waveform processing, 'ss' for spectrum to spectrum processing, 'ws' for waveform to spectrum, and 'sw' for spectrum to waveform processing. The function has the following signature:

```
function [s_out,user_config,state] = process_xy(s_in,...
                                                signal_dimensions,...
                                                user_config,state)
```

The parameters `signal_dimensions`, `user_config`, and `state` are described above. `wave_in` is a `InfxInf` array of doubles or complex numbers in the case of spectral processing.
The `release` function is used to do final cleanup if necessary. It takes no parameters and returns nothing:

```
function release()
...
end
```

3.2 User configuration

As mentioned before, user configuration must be initialized in the form of a vector of structs in the `init()` function. The elements of `user_config` may be changed during processing, but changes to `signal_dimensions` are not allowed. For every element of `user_config`, an openMHA configuration variable with the same name is created, allowing changes to the `user_config` in a real-time safe way. A current limitation is that changes made to `user_config` during `process()` are lost on configuration change from the parser side, so the `state` argument should be used to keep dynamic state like i.e. filter states.

3.3 State keeping

State keeping can be done in one of two ways. The quick option is to define the stateful variable as `persistent` or `global`. This has the downside that the variables are shared between all instances of the user library, making it unsafe to load the user library more than once at a time. The upside is that these variables may have any type and may change their size at any time, although changing their size may not be real-time safe.
Alternatively, the `state` input/output variable may be used as shown above in section 3.1.1. It has the same type as the `user_config` variable and follows similar rules. All elements of `state` must be initialized with *name* and initial *value* in `init()`. The size of the element `value` may only be changed during `init()` and `prepare()`. During `process()` only the elements of value may be changed, but not its size. Changing the size during `process()` is not real-time safe and leads to a crash. openMHA parses the contents of `state` and makes its elements available as monitor variables.

3.4 Deployment

In order to get a ready to use user library, the MATLAB Coder needs to be setup to use a compiler. Please refer to the MATLAB Coder documentation for how to do this. If the user library is compiled using a different compiler than openMHA there may be compatibility problems. If possible, use the MinGW compiler on Windows, the gcc compiler on Linux, and clang on macOS. If the MATLAB Coder can not be setup to use a compatible compiler, the generated code may need to be exported using the `packNGo` utility provided by the MATLAB Coder and compiled manually. For compilation, the same setup as is used to compile openMHA can be used. See `COMPILATION.md` for details.
In any case the user library (usually a shared library with the file ending `.so`, `.dylib`, or `.dll`)

then needs to be copied to where openMHA looks for its plugins. By default these locations are

- `C:\\Program Files\openMHA\bin` (Windows)
- `/usr/local/lib/openmha` (macOS)
- `/usr/lib` (Linux)

3.5 Example

This section describes step-by-step how to go from the empty template code in `examples/24-matlab-wrapper-simple` to a user library implementing a simple delay-and-sum algorithm, where the delay and the gain are real-time configurable on a per-channel basis. The finished code can be found in `examples/25-matlab-wrapper-advanced`.

Init

Let's take a look at the contents of `init.m`. We know we want two configuration variables: The delay and the gain, so we need a vector of two structs:

```
function [user_config, state] = init(user_config, state)
user_config =[struct('name','delay', 'value',ones(1,1)); ...
              struct('name','gain','value',ones(1,1))];
end
```

The first element of `user_config` is named `delay`, the second one is named `gain`. The actual value of the configuration variable is stored in the `value` member. As we want one entry per channel but do not yet know the number of input channels we just leave the initial value a 1×1 matrix of ones. Note that because of the fixed interface, `value` must always be a matrix of doubles, even if, as in this case, we only want to support integer values for the `delay` configuration variable. `state` is not used in this example.

Prepare

The next function of interest is `prepare()`, found in `prepare.m`:

```
function [signal_dimensions, user_config, state] = ...
prepare(signal_dimensions, user_config, state)
if(signal_dimensions.domain~='W')
fprintf('This plugin can only process signals in the time domain. ...
         Got %s\n',signal_dimensions.domain);assert(false);
end

% Need one delay entry per input channel
user_config(1).value=zeros(signal_dimensions.channels,1);

% Need one gain entry per input channel
user_config(2).value=zeros(signal_dimensions.channels,1);

% Number of output channels is always one
signal_dimensions.channels=uint32(1);
end
```

The first thing we do in lines 3 to 5 is to check `signal_dimensions` if the input signal we get is really in the waveform domain and if not print an error message and quit. Next we need to resize the delay and the gain to the appropriate sizes. Both are set to be vectors containing one element per channel. The number of channels is available as `signal_dimensions.channels`. As our user code changes the dimensions of the signal we need to announce this fact to the openMHA framework. We do this by changing the `channels` member of `signal_dimensions` to one. Note that in line 15 we need to explicitly cast the value to the appropriate type lest we get errors during code generation. Also observe that the change to `channels` was the last thing we did, as we needed the original value before!

process_ww, process_ss, process_sw and process_ws

In order to be able to use `make.m` for every combination of of input and output signal domain with minimal changes, we introduce `process_ww`, `process_ss`, `process_ws`, and `process_sw`. These entry-point functions serve as proxy for the real processing processing function and all contain the same code forwarding the call to the 'real' processing function:

```
1  function [s_out,user_config,state] = process_ss(s_in,...
2                                      signal_dimensions,...
3                                      user_config,state)
4  % This function provides the entry point for spectrum to spectrum
5  % processing and forwards the actual processing to process(...)
6  % Do not touch unless you know what you are doing!
7  % The actual processing should be done in process.m
8  [s_out,user_config]=process(s_in,signal_dimensions,user_config);
9  end
```

This allows us to keep the code invoking the MATLAB mostly the same. More on that later.

Process

```
1  function [wave_out,user_config,dummy] = ...
2  process(wave_in,signal_dimensions, user_config, dummy)
3
4  delay=user_config(1).value;
5  gain=user_config(2).value;
6
7  persistent state;
8  if(isempty(state))
9      state=zeros(signal_dimensions.fragsize+uint32(max(delay(:))),...
10                 signal_dimensions.channels);
11 end
12
13 persistent read_idx;
14 if(isempty(read_idx))
15     read_idx=uint32(zeros(signal_dimensions.channels));
16 end
17
18 persistent write_idx;
19 if(isempty(write_idx))
20     write_idx=delay;
21 end
```

```
22
23 for fr=1:signal_dimensions.fragsize
24     for ch=1:signal_dimensions.channels
25         write_idx(ch)=mod(write_idx(ch),...
26                           (signal_dimensions.fragsize+delay(ch)))+1;
27         state(write_idx(ch),ch)=wave_in(fr,ch);
28     end
29 end
30
31 wave_out=zeros(signal_dimensions.fragsize,1);
32 for fr=1:signal_dimensions.fragsize
33     for ch=1:signal_dimensions.channels
34         read_idx(ch)=mod(read_idx(ch),...
35                          (signal_dimensions.fragsize+delay(ch)))+1;
36         wave_out(fr)=wave_out(fr)+...
37                      state(read_idx(ch),ch)*10^(gain(ch)/10);
38     end
39 end
40 end
```

In lines 4 and 5 we define shorthand notations for delay and gain. This makes it easier to follow the code. Next we define some helper variables. We implement the delay line as a ringbuffer with the lag between read and write index appropriately chosen for the delay. As we want to delay every channel independently we need to keep state for every channel. The state vector needs to be able to contain all incoming samples of a block in addition to the delayed samples from the past. Because we need to keep the state in between calls to process, we define the state matrix and the read and write indices as persistent. Alternatively we could use entries in `state` to store them, as used in example 28 (`examples/28-matlab-wrapper-spec2wave/`). Next, we loop over the input signal to fill our state vector, advancing the write pointer appropriately. In line 31 we initialize the output signal to zero and then loop over the state vector, adding the delayed samples from different channels together.

Release

The last function we can fill is `release()`:

```
1 function release()
2 end
```

We do not need to do any cleanup, so we leave it empty.

Deployment

To generate the C code from our MATLAB code we can use `make.m` unchanged. Let's take a look:

```
1 function make(iodomain,varargin)
2 ...
3 end
```

The `make()` function takes several arguments, of which the first, `iodomain` is mandatory. `iodomain` determines the input and output domains of the generated code. It is a string and must take one of the following values:

ww wave to wave

ss spectrum to spectrum

ws spectrum to wave

sw wave to spectrum

The optional parameters are

outputName (string) The name of the generated library

packOutput (logical) If the generated code should be packed in a self-contained zip file

In this example, we invoke the coder with

```
make('ww','outputName','example_25','packOutput',true);
```

At the beginning we initialize the `coder` configuration object to some sensible defaults:

```
%% Create configuration object of class 'coder.CodeConfig'.
cfg = coder.config('dll','ecoder',false);
cfg.GenerateReport = true;
cfg.ReportPotentialDifferences = false;
cfg.SaturateOnIntegerOverflow = false;
cfg.FilePartitionMethod = 'SingleFile';
cfg.EnableOpenMP = false;
cfg.RuntimeChecks = true;
cfg.SupportNonFinite = false;
cfg.HardwareImplementation.ProdHWDeviceType = ...
                    'Generic->32-bit x86 compatible';
cfg.HardwareImplementation.ProdLongLongMode = true;
cfg.HardwareImplementation.TargetLongLongMode = true;
```

We define the target device, enable run-time checks, etc. A complete list can be found in the MATLAB Coder documentation. Changing the settings should only be done after careful consideration and never without reason.
The second section defines the types of the input arguments for `init()`:

```
%% Define argument types for entry-point 'init'.
ARGS = cell(4,1);
ARGS{1} = cell(1,1);
ARGS_1_1 = struct;
ARGS_1_1.name = coder.typeof('X',[1 Inf],[0 1]);
ARGS_1_1.value = coder.typeof(0,[Inf Inf],[1 1]);
ARGS{1}{1} = coder.typeof(ARGS_1_1,[Inf 1],[1 0]);
ARGS{1}{1} = coder.cstructname(ARGS{1}{1},'user_config_t');
ARGS{1}{2} = ARGS{1}{1};
```

Changing any of these required customization of the matlab_wrapper plugin. If such customization seems necessary choosing the native compilation approach is recommended.
The last section actually invokes the MATLAB coder:

```
codegen('-config',cfg,...
    '-o',outputName,...
    'init','-args',ARGS{1},...
    'prepare','-args',ARGS{2},...
```

```
    process_name, '-args',ARGS{args_idx},...
    'release');
```

The `-config` argument takes the configuration object we constructed earlier, `-o` takes the output file name, which can be passed as argument, otherwise defaults to the name of the first entry point function. The rest of the invocation defines the entry-point functions and their argument types pairwise. `process_name` and `args_idx` are determined by `iodomain`. If the generated code needs to be compiled by hand, we need to pass `packOutput` as `true` to execute the last block in `make.m`:

```
%% Optionally package the code for deployment elsewhere
if(packOutput)
    buildInfo=load(['codegen/dll/' outputName '/buildInfo.mat']);
    packNGo(buildInfo.buildInfo,'fileName',[outputName '.zip']);
end
```

This makes the `packNGo` utility compress all source code needed to compile the plugin into `outputname.zip`. We then can move the contents of the resulting zip file into a separate directory and compile them using Makefile provided in the example directory. Note that this also enables us to use the MATLAB Coder on one machine and deploy the generated code on another machine where the Coder is not available.
The makefile needed to compile the example is relatively simple as far as makefiles go:

```
example_25.so: example_25.o example_25_emxAPI.o example_25_emxutil.o
    $(CC) -shared -o $@ $^
```

In the first line we define the target library we want to compile and its dependencies: `example_25.so` depends on `example_25.o`, `example_25_emxAPI.o`, and `example_25_emxutil.o`. The Make program automatically provides the rule to how compile `.o` files from `.c` files. Here the first file, `example_25.c`, contains the actual generated source code, `example_25_emxAPI.c` and `example_25_emxutil.c` contain utility code handling the structures the MATLAB coder generates to handle variable size matrices and similar constructs.
This Makefile should be fairly universal and be usable for any system needing only small adjustments like e.g. the file suffix for shared libraries in and the actual name of the input and output files. If needed the compiler flags can be customized further. See the documentation of your compiler for details.
If, on the other hand, the MATLAB Coder was set up to use a compiler that produces openMHA compatible output we can just move the resulting user library, usually a shared library with the file ending `.so`, `.dylib`, or `.dll`, to the appropriate directory, start openMHA, point the matlab_wrapper plugin to the library and configure the user algorithm, like shown in `example_25.cfg`:

```
nchannels_in = 2
fragsize=128
srate = 16000

iolib = MHAIOFile
io.in = example_25.wav
io.out = out.wav

mhalib = matlab_wrapper
mha.library_name=example_25
cmd=prepare
mha.delay=[50 100]
```

```
13 mha.gain=[-5 -5]
```

The first two sections are the usual setup routine for openMHA where signal parameters like the number of input channels, the fragment size and the sampling rate are set up. We also tell openMHA to use file-to-file processing for this example with `example_25.wav` as input and `out.wav` as output, respectively Lines 9ff contain the actual configuration of the matlab_wrapper plugin. We first tell openMHA to load the library named `example_25`, leaving out the file suffix. We then need to issue the `prepare` command because we reset `delay` and `gain` to default values in our `prepare()` function, overwritting all values we would assign before the prepare call. Finally we set the delay to 50 and 100 samples respectively and the gain to $-5\,\mathrm{dB}$ for both channels.

4 Native compilation

If the code can not be rewritten to fit the wrapper plugin restrictions or when only parts of the algorithm shall be implemented in MATLAB the 'Native compilation' approach can be used. Here the user takes a the source code of a skeleton openMHA plugin and writes their own plugin, using the generated code only as building blocks, finally compiling the plugin as any other self written openMHA plugin. This approach is much more flexible but requires more interaction on part of the user. No step by step guide can be given, instead there are only some guidelines and examples to observe.

4.1 User configuration

The native compilation does not provide a ready made way to pass configuration parameters to the plugin. One way is to define the configuration as input arguments to the matlab function. The user then has to manually add MHAParser::* configuration variables and translate them to appropriate types and pass them to the generated code when calling the signal processing functions. Please the `examples/23-matlab-coder` for a beginner's example. This example can be adjusted for the end user's needs.